A Cooperative Botnet Profiling and Detection in Virtualized Environment

Shun-Wen Hsiao*†, Yi-Ning Chen*, Yeali S. Sun* and Meng Chang Chen†

*Department of Information Management
National Taiwan University, Taipei, Taiwan 10617
Email: {r93011, r99725028, sunny}@im.ntu.edu.tw

†Institute of Information Science
Academia Sinica, Taipei, Taiwan 11529
Email: {hsiaom, mcc}@iis.sinica.edu.tw

***Abstract*—Cloud security becomes an important topic in recent years, as to overcome the botnet in a virtualized environment is a critical task for the cloud providers. Although numerous intrusion detection systems are available, yet it is not practical to install IDS in every virtual machine. In this paper, we argue that a virtual machine monitor (VMM) can support certain security functions that our proposed design can actively collect information directly from the VMM without installing an agent in the guest OS. In addition, bot could not aware of the existence of such detection agent in the VMM. The proposed detection mechanism takes both passive and active detection approaches that the passive detection agent lies in the VMM to examine the tainted data used by a bot to check against bot behavior profiles and the active detection agent that performs active bot fingerprinting can actively send specific stimulus to a guest and examine if there exists expected triggered behavior. In the real-world bot experiments, we show the passive detection agent can distinguish between bots and benign process with low false positive and false negative rates. Also, the result shows the active detection agent can detect a bot even when before it performs its malicious jobs. The proposed mechanism suites an enterprise having cloud environment well to defeat malware.**

***Keywords*—bot, fingerprinting, hypervisor, intrusion detection.**

I. Introduction

Botnet is a network that consists of massive malware-infected hosts (i.e., bots), controlled by a botmaster and exploited to launch a large-scale cyber attacks. The botmaster uses certain specific network protocol (e.g., IRC, HTTP and P2P) as the Command and Control (C&C) channel for delivering malicious executables or instructions to the bots. For botnet detection, researchers have analyzed the behavior of a botnet and discovered specific patterns commonly shown in a bot lifespan. Rajab et. al. [1] discovered that the common operation patterns of an IRC botnet are: exploit, bot download, DNS lookup, join, and command.

The understanding of the bot behavior helps us to distinguish an infected host from a normal host by detecting the specific behavior patterns. However, in order to evade such behavior detection, bots in the real world change their structures and complicate attack procedures to make their behavior harder to be analyzed, including:

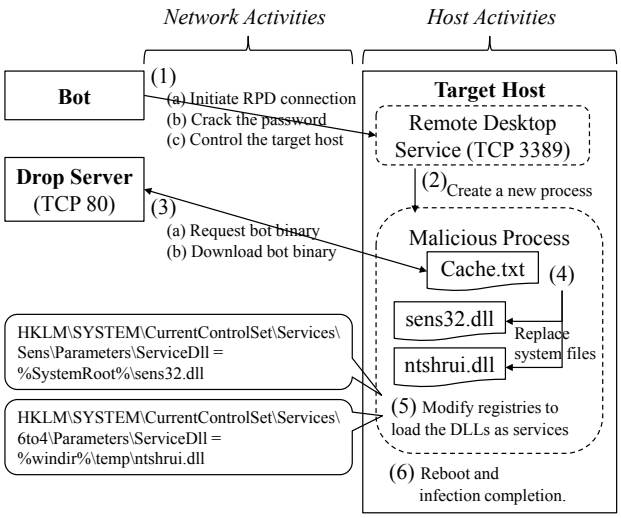

Fig. 1. The infection procedure of bot W32.Morto.A.

- Change on the exploitation. For example, attacker cam make the exploitation more stealthy through phishing or social engineering.
- Change on the structure of DNS. Using the technique of fast-flux to hide C&C servers and other bots.
- Change on the communication. Use a self-defined or encrypted protocol to prevent from being analyzed.
- Change on structure. For example, a botnet structure may change from the centralized botnet to a decentralized P2P structure.

Fig. 1 is a bot example `W32.Morto.A`, found in late 2011. We collected the bot binary from infected hosts and reproduced the infection process in our lab. Fig. 1 shows its behavior that it (1) exploits a vulnerable host, (2) creates a temporary malicious process, (3) obtains the bot binary from network, (4) replaces benign system files, and then (5) makes itself as a resident service (6) after next system boot up.

Through the observations, bot behavior can be generally divided into two categories: network activities and host activities. Correspondingly, there are two popular detection approaches: network-based and host-based detection approach. The former one targets on network activities that can be observed from the network; while the latter one focuses on the information that can be collected directly from the host.

Network-based detection approach often implements as an intrusion detection system (IDS) to monitor the network traffic by signature matching or anomaly detection techniques. It has the advantage of monitoring massive hosts at the same time. Also, it is transparent to the monitored hosts, which means the hosts (or even the bots) are not aware of the existence of such detector. However, as a passive detector, it may miss some malwares when they are in their incubation period (e.g., with no network activities) or when they make stealthy communication. Such limitation of a passive detector may lead to a less effective detection.

A general user of personal computer usually selects host-based IDS. It is a software installed at the end host, e.g., anti-virus. Similarly, the installed agent can adopt signature matching or anomaly detection technique to detect suspicious processes or files at the host. For detecting the existence of bots, it often targets on the following entities at the end hosts [2], [3]: (1) the files that the malware created or modified; (2) the Windows registry entries that the malware created, modified or deleted; (3) the DLLs loaded by the malware before executing; (4) the network connection established by the malware and its context; and (5) other information, e.g., the service and kernel modules installed by the malware.

A host-based detection system can closely monitor the actions inside the host and collect extra information than a network-based detection system. However, the detection agent may be detected by the bots and it can only monitor the very host that installs it. Once a malicious software is aware of the existence of detection agent, it may hide or change its behavior to evade the detection. If we need to guarantee that all of the host activities that we concern can be observed and logged, a transparent detection mechanism should be introduced.

Meanwhile, a growing number of enterprises start moving their services to virtualized environments (e.g., private cloud) for flexibility, mobility and cost reduction. The hypervisor supports virtual hardware for the guest OS and allows the isolation capability to virtual machines. However, there has not been studied of bot detection in such virtualized environment.

One more limitation of both detection approaches is their passivity. It means they must first observe particular bot activities and then trigger the security alarm. In other words, the passive detection system can only detect a malware after malicious packets or behaviors are observed by the detector.

In order to overcome (1) the passivity issue of current detectors and (2) the transparency issue of host-based detection approach, and to adopt the advantage of both network-based and host-based detection approaches, we propose a bot profiling and detection mechanism within the VMM with both passive and active detection approaches. One more benefit of the design is the saving from installing only one detection system in the VMM to simultaneously monitor all virtual machines. The goals and contributions of our work are as follows.

- Design and implement an intrusion detection system in the QEMU hypervisor.
- Implement a process tracking mechanism to trace log and profile the behavior of a specified bot process and all its spawn processes.
- Use known bot variants to populate system API related profiles for later active/passive detections.
- Compare behavior profiles of bot families and benign processes, and quantitatively measure their difference.
- Analyze the behavior patterns of bots and benign processes from the perspective of API calls.
- Develop a method to generate active fingerprinting for a bot family.

The main differences between our proposed system with the literatures are that (1) the implemented tracing mechanism adopts the concept of tainted analysis so that all the target processes, files, registries and memory blocks are monitored and traced. Therefore, the list of the monitored target will dynamically grow when a new process is spawn, a tainted file or registry is accessed by other process, a memory block is accessed/copied, and etc. (2) We develop our behavior profile from the perspective of system API calls. We can precisely portray the behavior of an attack to allow further analysis. (3) We propose the concept of active fingerprinting technique residing the VMM that it changes the paradigm of passively detecting the existence of malware. The administrator can actively probe the existence bot in a virtualization environment before a breakout.

The proposed passive detection agent lies in the VMM to examine and track the tainted data used by a suspicious virtual host (i.e., guest) and check it against the bot behavior profile. Such design can provide sufficient transparency to the monitored guests. Moreover, it can also monitor multiple guests on the same physical machine at the same time.

The proposed active detection agent that performs active bot fingerprinting can actively send certain specific stimulus (that is derived from the bot behavior profile) to a virtual host and examine if it is a bot by observing whether certain expected behavior is triggered by the stimulus. Our experiment shows that it can diminish the problem of passivity and is a good tool for network administrator to evaluate a guest without installing additional detection agents. Therefore, our proposed mechanism suits for an enterprise having virtualized environment (e.g., private cloud) well to defeat malwares and ensures the security of all the virtual hosts.

The remainder of the paper is organized as follows. In Section 2, we review some existing works of detection and fingerprinting. Section 3 is our proposed detection mechanism. We present the system design and discuss its implementation in Section 4, and evaluate our design with real-world bots in Section 5. Section 6 contains some concluding remarks.

II. Related Work

A. Botnet Detection

Several network-based approaches have been proposed such as BotHunter, BotSniffer and BotMiner. BotHunter [4] constructs a botnet infection dialog model by correlating the two-way communication, and uses the model to detect the intrusion activities. BotSniffer [5] detects bots within the same botnet since their activities have spatial-temporal correlation and similarity. BotMiner [6] performs cross cluster correlation to identify the host that shares similar communication patterns and malicious activity patterns.

Panorama [7] traces the information flow of predefined taint data in the host, and observes how and when a malware process leverages them. While BotTracer [8] detects malicious behavior through observing the process which invokes sensitive system call with the assistance of virtual machine technique, API hook tool and network monitoring tool. This approach requires the predefinition of normal application to filter out the process started by bot.

Past works usually detect botnet by focusing on bot or botnet malicious behavior, no matter on the network or in the host. In these cases, we anticipate that bots or bots within the same family indeed exist certain malicious behavior that can be used to distinguish from each other or a benign software. Therefore, we can make a profile for a bot as a fingerprint and use it to check against others.

B. Virtual Machine Introspection

Garnkel [15] proposed an architecture for intrusion detection using virtual machine introspection (VMI) technique. They provide six sample security policies and monitor them with a modified VMware Workstation. The ReVirt [16] targets on moving security logging mechanism into a virtual machine. Chen et al. [17] stated that secure logging and intrusion detection could benefit from the virtualized environment. VMwatcher [18] overcame the semantic gap challenge to reconstruct internal semantic views (e.g., files, processes, and kernel modules) of a VM. SIM [20] takes another approach to install an internal entities as well to have a semantic-rich view of the guest. A formal model [21] of VMI is even proposed for for describing VMI techniques. We anticipate that VMI research is a promising approach for monitoring and logging malware activities due to its isolation and transparent property.

The main difference of our proposed IDS mechanism is that we focus on the system API calls to describe the activities taken by the bots and we tracking all the behaviors from the viewpoint of tainted analysis. Every memory, file or registry that a process (and its spawn processes) access is monitored and traced. Based on the profiles, we can populate behavior-based signature for later use and even use it as an evidence for forensics. The most important of all, we use them for active fingerprinting, which has not been explored yet.

C. Fingerprinting

Comer and Lin [9] in 1994 first developed fingerprinting approach, using the different configurations of each OS's TCP protocol implementation (different default value of parameters such as TTL) to detect the OS version of the remote host. Today, fingerprinting technology has been widely used in detecting (1) OS Fingerprinting [10]: the OS version of remote host, (2) application fingerprinting [11]: application version run in the remote host, and (3) vulnerability fingerprinting: whether a known vulnerability existing in the host.

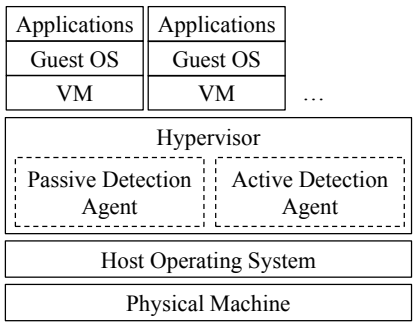

Fig. 2. A virtualized environment and the proposed agents.

To detect the OS version, for example, active fingerprinting approach initiates a special-designed packet, and detects the OS version of a remote host depending on the contents of response packet. On the contrary, passive fingerprinting approach dose not send any packets, but monitors packets inbound and outbound the host to determine the OS version. In this paper, we adopt the concept of fingerprinting and introduce a profiling system for fingerprinting malware.

III. Detection Approach

A. Monitor the Guest OS in a Virtualized Environment

In a virtualized environment (Fig. 2), a guest OS is executed on a virtual machine created by the hypervisor/emulator. The I/O and the CPU instruction of the guest OS are all translated by the hypervisor/emulator, and then are actually executed by the physical machine. Therefore, we implement the proposed detection agents in the visualization layer to monitor all the interactions between the guest OS and the VMM. Such approach makes it possible to monitor the guest OS without modifying it or installing additional software and reduce the risk of being detected by the malware in the guest OS. Another benefit is that the detection agents can monitor all above guest OSes at the same time.

However, the virtual machine introspection (VMI) approach must consider the semantic gap [8] between the OS-level semantics and low-level VM observations. We will demonstrate how to monitor the high-level activities by analyzing related data in the virtualization layer later.

B. Learning-Based Bot Behavior Profile

In a bot host, the initial bot process may create one or more processes to perform malicious activities. Our goal is to trace and characterize these bot processes' behavior and generate a bot behavior profile for them. We adopt a learning approach to generate the behavior profiles of real world bot samples.

The example of W32.Morto.A shows that a bot may access/modify specific files or Windows registries. We observe that certain actions are inevitable for the bot, hence we focus on building the bot's behavior profile by using the file/registry access activities related to the bot processes.

We define an *activity* as a system API call related to a file/registry access. The proposed passive detection agent will generate a *bot process activity log* that contains the activities of a set of monitored bot processes. Based on the collected bot process activity log(s) of a family of bot, we then generate a *bot behavior profile* that contains the common activities from each bot process activity log. Then, we may generate several bot behavior profiles based on different families of bots to create a *bot (malware) behavior database*.

In figure 3, we give a simplified example of the XML-based bot profile that we generate in our system. The bot is named W32.Virut and we use 10 different variants of W32.Virut to populate the bot behavior profile. The actually length of the profile is 206 lines, and we only list some of them. With this profile, we can easily describe the bot and use it as a basis to detect the bot in a hypervisor.

C. Passive Bot Detection

Based on the database, we identify a set of files and registries that are used to monitor and mark them *tainted*. For example, a malware that would like to run as a system service in the infected host can modify or add a Windows registry entry in the HKLM\SYSTEM\CurrentControlSet\Services\. Take W32.Morto.A for instance, it modifies a registry entry named 6to4. Therefore, it is marked as tainted.

Then, we check the processes to be tested in the guest OS against the tainted objects in the runtime. If any process accesses the tainted objects, it is marked suspicious. Then immediately, the passive detection agent starts to trace the activities of this suspicious process (and its spawn processes) for a period and generates the corresponding activity log.

The passive detection agent then analyzes the collected process access activity log against the bot behavior profile database. To determine the abnormality, we calculate the Jaccard similarity coefficient between a bot behavior profile and a process activity log to measure the similarity of them.

The tainted file Jaccard index, $\alpha_J(i,k)$, and the tainted registry Jaccard index, $\beta_J(i,k)$ are computed for a possible bot k in guest host i based on the process activity log L_i and the bot behavior profile L_k. They are defined as followings.

$$\alpha_J(i,k) = \frac{\text{\# of tainted file access activity in } L_i \cap L_k}{\text{\# of tainted file access activity in } L_i \cup L_k}$$

$$\beta_J(i,k) = \frac{\text{\# of tainted registry access activity in } L_i \cap L_k}{\text{\# of tainted registry access activity in } L_i \cup L_k}$$

If both values are 1.0, it implies that the guest host i is infected by the bot k since they show the exactly same activities. Proper thresholds α_k and β_k for each bot k are set for effective

```
<?xml version="1.0"?>
<Report>
 <Module>
  <LoadLibrary fileName="ws2_32.dll"/>
  <LoadLibrary fileName="wininet.dll"/>
  <LoadLibrary fileName="msvcrt.dll"/>
  ...
 <File>
  <CreateFile creationDisposition="
     OPEN_EXISTING" fileName="C:\WINDOWS\
     system32\vmwpfh.exe"/>
  <CreateFile creationDisposition="
     OPEN_EXISTING" fileName="c:\autoexec
     .bat"/>
  <CopyFile existingFileName="C:\malware.
     exe" newFileName="C:\WINDOWS\
     system32\vmwpfh.exe"/>
  ...
 <Registry>
  <RegQueryValue hKey="HKEY_LOCAL_MACHINE
     \System\CurrentControlSet\Services\
     Tcpip\Parameters\Hostname" type="
     REG_SZ"/>
    <RegQueryValue hKey="
       HKEY_LOCAL_MACHINE\System\
       CurrentControlSet\Services\Tcpip\
       Parameters\Domain" type="REG_SZ"/>
  ...
  <RegCreateKey hKey="HKEY_USERS\S
     -1-5-21-117609710-...\Software\
     Microsoft\Windows\CurrentVersion\
     Explorer\User Shell Folders"/>
  ...
  <RegSetValue hKey="HKEY_CURRENT_CONFIG\
     Software\Microsoft\windows\
     CurrentVersion\Internet Settings\
     ProxyEnable" type="REG_DWORD"/>
  ...
 <OtherAction>
  <OpenProcess desiredAccess="
     PROCESS_QUERY_INFORMATION
     PROCESS_VM_READ" procName="rundll32.
     exe"/>
  <CreateProcessInternal cmdLine="C:\
     WINDOWS\system32\vmwpfh.exe"/>
  <WinExec cmdLine="C:\WINDOWS\system32\
     vmwpfh.exe"/>
  <OpenProcess desiredAccess="
     PROCESS_CREATE_THREAD
     PROCESS_QUERY_INFORMATION
     PROCESS_VM_OPERATION
     PROCESS_VM_WRITE" procName="explorer
     .exe"/>
  <CreateRemoteThread procName="explorer.
     exe"/>
 </OtherAction>
</Report>
```

Fig. 3. The bot behavior profile of W32.Virut (simplified).

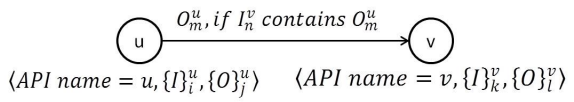

Fig. 4. Bot dataflow relationship diagraph.

detection, since bot variants might exist. Moreover, we might also not be able to observe or log all activities of a bot during a limited observation period. Therefore, if $\alpha_J(i,k) \geq \alpha_k$ and $\beta_J(i,k) \geq \beta_k$, the host i is determined as a bot infected by k.

D. Active Bot Fingerprinting

We observe that bot may have certain hidden behavior which is activated only when it is properly triggered (named trigger-based behavior [12]). For instance, a bot may launch an attack only after receiving a command from its bot master.

In the past, bot detectors can only passively detect bots after the bot shows certain malicious activities. We believe the trigger-based behavior can be a niche. Hence, we target on them and develop active bot fingerprinting, which is derived from OS fingerprinting concept. We analyze the bot behavior profile established in the previous phase and try to extract a bot fingerprint in this phase.

For each bot behavior profile in the database,

1) We list the I/O-related activities invoked by the bot process in terms of API calls with the input parameters and output results. Note that in a bot behavior profile, there are certain general APIs, files or registries (such as invoking `wsock32.dll` or querying registry `netshell.dll`). Hence, in our experiments, we manually exclude them. In practice, we can learn the exclusion list from benign processes. We will discuss it in our experiments later.
2) From 1), construct the bot dataflow digraph (Fig. 4) where a vertex is an API call. If there exists a directed edge from vertex u to vertex v, it means the intersection of the output value set of this instance u ($\{O\}_j^u$) and the input value set of this instance v ($\{I\}_k^v$) is not empty and the timestamps of this instance u is earlier than that of v. We set the value of the edge to O_m^u. It indicates that certain value (O_m^u) is passed from this instance of u to v.
3) From 2), generate a fingerprint which contains stimulus and response in pair. The stimulus is set to the API call u and the response is set to the API call v. If we change the value of o_m^u and theoretically it will be passed to v. If u is a triggerable event, then we shall be able to perform such active fingerprinting.
4) From 3), we perform the active bot fingerprinting through the stimulus, and then observe if the expected response (i.e., v) is triggered with the changed parameter. If the triggered response is observed, we consider the guest host as infected.

In our experiments, a triggerable I/O event is most likely a file change, a registry change or network packet received.

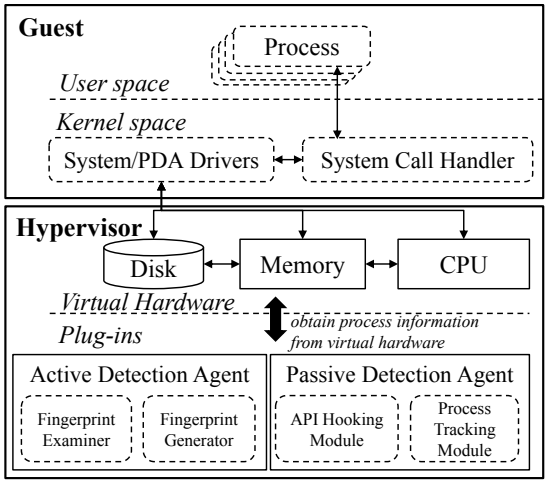

Fig. 5. The proposed system overview.

IV. System Design and Implementation

The proposed system (see Fig. 5) consists of passive detection agent (PDA) and active detection agent (ADA). These agents are implemented in the hypervisor/emulator as a plug-in. For Windows, a PDA driver needs to be installed in the guest OS to deliver the upper layer's information (e.g., processes information) to the underlying agents. Note that for Linux, such driver is not needed.

A. Passive Detection Agent

Process Tracing Module. Before tracing a process, PDA acquires the process information of the guest OS sent by the PDA driver. The returned process information includes: process name, `PID`, `CR3` value (each process/thread has its own `CR3` register value for mapping the virtual memory to physical memory), and the information of loaded modules.

In Fig. 6, the process tracing module gets the current executing process and current `CR3` value (step 1). We can instruct it to monitor a target process by using `CR3`, `PID` or process name. If the target process is executing (step 2), the current status of CPU and memory can be retrieved for further inspection (step 3). Further, we can call the API hooking module to record the process activities (step 4).

API Hooking Module. In our system, we monitor the bot behavior by API hooking, which is a common approach for detectors [3]. Traditional API hooking technique modifies the API function's address in DLL file and points the new address to a self-defined function to intercept the call. However, the malware may be aware of the presence of such API Hooks.

Our approach is stealthier, because our agent can obtain the status of virtualized hardware. In our system (Fig. 7), we monitor the value of `EIP` register to check which API function is called (step 2). If it matches a target API call, a self-defined callback function will be invoked in the VMM (step 3). We can then obtain the inputs and outputs of the call from the corresponding memory and `EAX` register (step 4–7). The hooked APIs are listed in Table I.

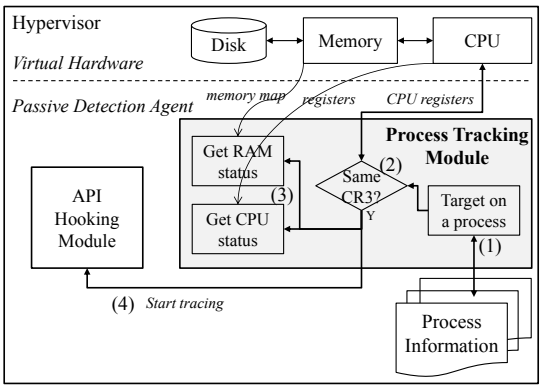

Fig. 6. The process tracing module in PDA.

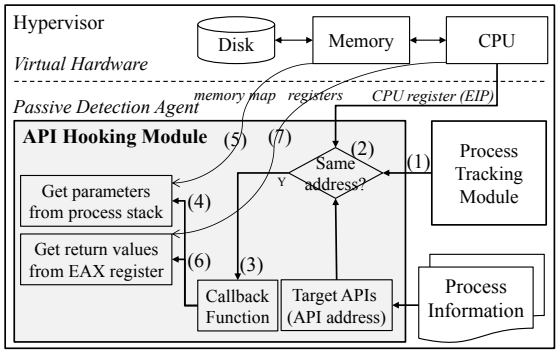

Fig. 7. The API hooking module in PDA.

B. Active Detection Agent

Active Detection Agent contains a fingerprint generator and a examiner. Generator generates bot fingerprinting and the examiner instructs the ADA driver to initiate the stimulus of a fingerprint. The examiner checks if the expected response is triggered. It can be performed periodically to check the bot infection situation, even when a bot is in its incubation period.

C. Implementation

We implement our system based on TEMU [13], a dynamic taint tracing platform built upon QEMU [14]. QEMU is an open source generic machine emulator and virtualizer. QEMU provides a system mode that can emulate a full computer system. TEMU includes a taint analysis engine and a semantics extractor. It can perform dynamic taint analysis, OS awareness, and in-depth behavioral analysis (e.g., memory accessed and API calls invoked by a process).

TEMU provides an API to extend its functions. We implemented the PDA and ADA in C/C++ as plug-ins of TEMU (Fig. 8). A Xerces-c Library is used to generate and parse bot behavior profile in XML format. In our experimental environment, the host OS is an Ubuntu 10.10 and the TEMU version is 1.0 (QEMU 0.12.5). To accelerate the emulated CPU, we used the accelerator kqemu 1.3.0pre11. We installed Windows XP SP3 as our guest OS as the testing environment since it is one of the most popular OS.

TABLE I. HOOKED APIS

File Access (kernel32.dll)	Registry Access (advapi32.dll)	Others
CreateFile	RegOpenKeyEx	LoadLibrary
ReadFile	RegQueryValueEx	OpenProcess
WriteFile	RegSetValueEx	CreateProcess
DeleteFile	RegCreateKeyEx	CreateProcessInternal
CopyFile	RegDeleteKeyEx	WinExec
	RegDeleteValue	ExitProcess

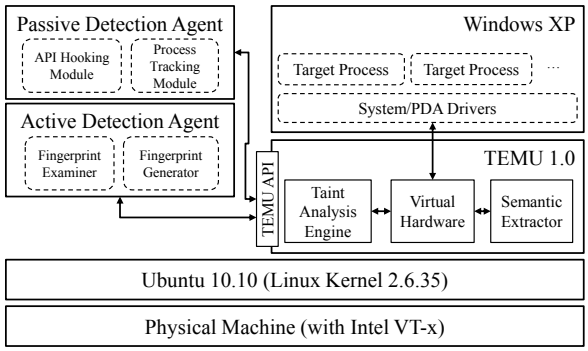

Fig. 8. System implementation.

Beside the new plug-ins, we also implement certain new features in QEMU or TEMU, such as (1) tainted process tracing that can automatically record and trace the new processes created by the bot process and its child process, (2) API parameters retrieval that can obtain the value of function parameters and return value for Windows API hooking, and (3) a subsystem for event logging.

Event Logging Subsystem. A `BlockDriver` is used to log the differences between a clean system (a.k.a. base file) and the malware-infected system. In QEMU, it is called backing file. It will not be modified unless the administrator commit it. The backing file can help us the verify our detection result. Moreover, for the real-time event monitoring, once a hooked API is found, the information of the corresponding parameters and process will be delivered to the standard I/O or file.

Tainted Process Tracing. A malicious process may write certain malicious materials into file system, and then the tainted files or directories may read by another process again. In order to keep tracking such activities, all the files that a malicious process writes to the file system should be tracked. In QEMU, we have to check such tainted file list and see if there is any propagation made by such malicious processes. We record the `PID` and `CR3` of the process to keep such tracking in an internal list. A corresponding mechanism for updating `CR3` value of current CPU and a list for maintaining current tainted file list are also implemented. We also implement a function to obtain the tainted file name from file system `inode` for convenience. If there is any access to these tainted files, an alarm is triggered. A function of TEMU plug-in, `tracing_taint_disk` ,is modified to accomplish this job.

We add a new data structure named `trace_profile_t` to keep the tracking the behavior of a process into a profile. It includes the process name in the file system, a general log file, a network activity log file, an API hook log file, a memory allocation log file, the `PID` and `CR3` of the process. All the

tracing control commands (such as start tracing, stop tracing, set the PID to be traced, set the process name to be traced, output the profile, output our log files) can be used in QEMU's standard command line interface.

API Hooking. In order to monitor the APIs in Table I, we register these APIs in a list named hooks. For all these Win32 APIs, they are either from kernel32.dll or advapi32.dll library. We then further specify the structure of the parameters and return values of these API functions from Windows MSDN references [19]. Hence, after the API is called, we can obtain the parameters and return values.

Take the CreateProcessInternalW API for example, it has 11 parameters and returns a BOOL. After the API is called, we first intercept the call and read memory stack starting at ESP with size 32-bit * 12 (11 parameters and 1 return value). All these information will be copied to another memory and wait for further logging. Actually, we implemented 43 API hooks for these APIs in Table I because there are several variants of a call. For example, CreateProcessA, CreateProcessW, CreateProcessInternalA, and CreateProcessInternalW are all possible API calls to to create a process. The C++ file for implementing the hooking functionality is 2239 lines.

V. EXPERIMENT

A. Data Source

The bot samples are downloaded from a database built by the National Center for High-Performance Computing (NCHC), Taiwan. We choose four families of botnet in our experiments: Virut, Sality, Korgo and Pinfi. The bot family classification is provided by Symantec Security Response and VirusTotal by using the hash value of the bot binaries. For each bot family, we have 10 variants (V_1 ... V_{10}). They are the largest family in the database. We use Windows Internet Explorer (IE) as an example of benign process. In some experiments, Chrome, MSN Messenger (MSN) and Microsoft Paint (Paint) are used as well.

B. Learning-based Bot Behavior Profiling

We build the bot behavior profile from bot process activity logs. We infect vulnerable guest OS by using bot samples, and instruct the passive detection agent to monitor and log the bot-related processes (including the bot process and the processes it creates or injects malicious code to) for 2 minutes.

For generating the behavior profile for a bot family, we use multiple variants and extract common entries form their bot process activity logs. It is expected that the different set of chosen variants could populate different profiles since the bot variants are slightly different from each others. We will show how it affects the detection later. We also expect different bot families should exist certain distinct behavior. Table II shows the number of file/registry accessed in our experiment. As we expected, difference bot families have different characteristics. We discuss certain interesting APIs, files and registries here.

Process-Related APIs. CreateProcessInternal, WinExec, OpenProcess, and CreateRemoteThread fall into this category. It is quite common to spawn other new processes to perform additional malicious jobs. On the contrary, IE only opens new process named IEXPLORER.EXE, which is exactly itself. For Korgo, it uses WinExec with command line parameter: C:\WINDOWS\system32\zaegr.exe to execute a bot. Alternatively, Virut uses OpenProcess to invoke rundll32.exe, which is responsible for running dynamic link library in Windows system.

For creating process, an system administrator could set a limit to the number of spawn process. It could be helpful for counteracting DDoS-like attacks, worm or bot, since some of them create lots of process in a very short period of time.

CopyFile and DeleteFile. All of our bots use CopyFile to make a copy of the bot binary to Windows system folder or temporary folder for further execution. The filenames may look like zaegr.exe or vwjop.exe. Sality deletes several files, and they are all temporary files created by itself. For IE, it only deletes HTTP cookies and temporary HTML files in IE's temporary folder.

CreateFile. CreateFile API has a parameter named creationDisposition, and it could be CREATE_NEW, CREATE_ALWAYS or OPEN_EXISTING. From our observation, bots usually use the former two parameters to create files (in the system folder) for later use and use the latter one to read/execute existing files. But for IE, it uses OPEN_EXISTING for reading cookies, cached HTML and font files (from its temporary folder and font folder).

From the viewpoint of files, it is much secure to match signature in the VMM, since malware may detect the existence of IDS as well. A host-based IDS might be disabled by a malware, but it could not disable an IDS in the VMM. In addition, if a new tainted file is identified, the administrator can add it in our QEMU/TEMU plug-in to actively check the infection status of all guest OSes. It would be more efficient.

RegCreateKey and RegSetValue. Bot may modify or add registry to change the behavior of the infected host, such as adding new service (all bots), change hostname/domain (Korgo), change firewall settings (Sality), disable Limited User Account (Sality), and etc. The further forensics should be done; however it may beyond the scope of this paper.

Files and Registries. Table III shows the average Jaccard similarity coefficient (and its standard deviation) between every bot variants within its family, and the coefficient comparing with IE. The similarity within the bot family is higher than the similarity of a bot and IE. From the point of view of registry, Sality variants have lots of common behavior (.3979), but the difference (.1910) between variants is also large.

We observe (1) the number of accessed file is far less than that of registry, which suggests detecting malicious file is important but may have higher false positive if the detectors miss them; (2) only very few files have common filenames among variants since random filename are used; (3) bots usually do not use random registry; (4) variants in the same family have common behavior and using known variants may detect unknown ones; (5) IE is very different from these bots.

TABLE II. THE NUMBER OF API CALLS OF COLLECTED BOT FAMILIES AND THE BENIGN PROGRAMS

	File			Registry			Others	
	Create	Copy	Delete	QueryValue	CreateKey	SetValue	LoadLibrary	Others
Korgo	8.6	1.0	0.0	131.3	12.0	10.7	24.0	5.9
Pinfi	16.5	1.2	0.1	189.6	19.5	20.2	25.9	6.5
Sality	12.3	0.8	2.3	466.3	18.1	388.0	23.0	13.6
Virut	7.0	1.0	0.0	119.1	16.7	14.4	23.8	5.2
IE	163.0	0.0	8.0	905.0	34.0	39.0	40.0	3.0
Chrome	93.0	1.0	10.0	409.0	94.0	8.0	46.0	4.0
MSN	14.0	0.0	0.0	350.0	19.0	19.0	19.0	0.0
Paint	4.0	0.0	0.0	188.0	21.0	0.0	11.0	0.0

TABLE III. THE SIMILARITY COEFFICIENT OF BOT VARIANTS USING FILE AND REGISTRY RELATED API.

	Korgo	Pinif	Sality	Virut
Family Member (File)	(.1879, .0666)	(.0131, .0466)	(.1107, .0627)	(.2269, .0657)
Family Member (Registry)	(.2629, .0690)	(.2050, .0916)	(.3979, .1910)	(.2320, .0654)
with IE (File)	(.0323, .0132)	(.0394, .0154)	(.0251, .0138)	(.0365, .0119)
with IE (Registry)	(.0949, .0090)	(.0854, .0177)	(.0392, .0193)	(.0774, .0228)

TABLE IV. THE NUMBER OF TAINTED FILE/REGISTRY ACCESS AGAINST DIFFERENT BOT PROFILE

	$\|P\|=1$	$\|P\|=2$	$\|P\|=3$	$\|P\|=4$	$\|P\|=5$
Sality (File)	15.45	3.25	2.20	1.50	1.25
Virut (File)	8.00	5.40	5.05	5.00	5.00
Sality (Registry)	1218.20	803.15	617.05	447.65	226.65
Virut (Registry)	182.35	133.95	130.50	129.55	128.65

TABLE V. THE JACCARD COEFFICIENT RESULT OF VIRUT PROFILE USING DIFFERENT VARIANT SET

	P=1	P=2	P=3	P=4	P=5
$\alpha_J(V_i, Virut)$	.2269	.2509	.2777	.2981	.2549
$\alpha_J(IE, Virut)$	.0365	.0364	.0455	.0378	.0434
Diff	.1903	.2145	.2322	.2603	.2116
$\beta_J(V_i, Virut)$	.2320	.2506	.3050	.3307	.3057
$\beta_J(IE, Virut)$	.0774	.0898	.0930	.0925	.0952
Diff	.1546	.1608	.2120	.2382	.2105

C. Passive Bot Detection

1) Bot Behavior Profile Generated by Multiple Bot Variants.: We would like to understand what is the proper number of variants we should use in our proposed profile mechanism and how it affects our detection result. For a bot family, we infect the vulnerable OS by using variants in set B one by one and check against the accessed tainted file/registry with the bot behavior profile generated by set P (where $|B| + |P| = 10$, since for each bot family we have 10 variants). We randomly distribute these variants to B and P set, and repeat the experiments 20 times to obtain the average. We use `Sality` and `Virut` as demonstrative examples in Table IV.

As we expect, including more variants to generate bot profile could make it be more prune (but we may lose particular behavior for a specific variant). For `Virut`, the average number of registry for $|P| = 5$ is 128.65, which means `Virut` variants have commonly shared registry access behavior. However, the differences between `Sality` variants are large (the commonly shared registries decrease from 1218.20 to 226.65). It also matches the larger variance value, 0.1910, in Table III). Hence, making proper bot profile for `Sality` is not easy. We suggest further classification to `Sality` family.

We discover some registries are common network registries. Since checking the network configuration and establishing network connection are common for a bot and a benign network program, we suggest that it is necessary to introduce a white list for commonly used registries to increase the quality of the profile. As for the files, certain files are important and commonly used by the family, such as malware binaries. They could be useful for anti-virus detectors.

2) Jaccard Similarity Coefficient.: We use the Jaccard Similarity Coefficient to distinguish bot process and benign process. In the $P = 4$ experiment, for bot k = `Virut`, the Jaccard difference value of $\alpha_J(V_i, k)$ and $\alpha_J(IE, k)$ is 0.2603, and the difference value of β_J is 0.2382. The larger difference, the more we can distinguish a benign process and a bot process. Due to the page limit, we only show the result of `Virut` in Table V. We can see that the best result is at $P = 4$. In our experiments of other bot families, the minimal difference for β_J (register) is 0.1127 and the minimal difference for α_J (file) is 0.0498. These values can be used as thresholds to detect bots. We believe it is large enough for a IDS to distinguish a benign process from a bot process. It also indicates that usually using registry information as profile for detecting malware is much more efficient. Moreover, using proper number of variants to generate profile is important, too.

D. Active Bot Detection

In the active fingerprinting experiment, we generated an active fingerprinting for `Virut` based on its profile of 5 variants. The stimulus used here is to set two registries under directory `\HKLM\System\CurrentControlSet\Services\Tcpip\Parameters`. The first registry entry is `Domain` and the other is `Hostname`. The former one is often set to `NULL` for a normal host when the computer does not join a domain, while the latter one is usually the name of local host. `Virut` infects the vulnerable host by allocating and writing memory in the address space of `Explorer`. (Note `Explorer` is a user shell of Windows, not IE.) `Virut` then creates a remote thread that runs in the address space allocated from `Explorer`. Further, it queries these two registries and makes a DNS query to the domain specified in the registry. In this case, the DNS query is the response of our fingerprinting.

We test this fingerprinting on all the `Virut` variants, and observe that all the variants access these two registries and send the DNS packet out, except one variant accesses them but does not send the DNS packet out. To our best guess, it either delays the query or the attacker rewrites the code. For this specific active fingerprinting, all benign programs (i.e.,, IE,

Chrome, MSN, and PAINT) do not response it. While tracing a `Virut` variant, we observe that it spawns 4–6 processes (with different PIDs) to perform the attack. Hence, it is necessary to keep tracking the process and API calls. The result shows that it is an effective detection approach with no false positive.

VI. Discussion and Conclusion

Several measures are introduced in the literature to evaluate the similarity between two event sets. In this paper, we only take the existence of events into account, but we also anticipate the order of the activities might be meaningful for profiling. Therefore, time information is preserved in our design for further investigation in our future work.

For zero-day malware detection as a future work, we believe it can possibly come up with certain file and registry infection rules which are generalized from different kind of bot families. Such detection rules can be used in sandbox or honeypot to see if our method can catch zero-day malware.

As for network-related APIs, such as `AcceptEx` and `ConnectEx`, we argue that analyzing network protocol needs to reassemble multiple packets and reconstruct the communication context statefully. For the packet inspection and reconstruction, network-based approach, such as our previous work [22], is much more appropriate and easier for implementation, while it can keep the state of the protocol and infer the attacks by checking the integrity of the protocol execution context.

In Table I, although there are only 17 APIs are hooked (which means we only use them to profile a malware), yet the parameters of the APIs are also considered while comparing two processes. We believe the APIs and the parameters used are descriptive enough to identify these malware. However, we can also anticipate that a sophisticated attacker may try not to use Win32 APIs to build their malware. But we argue that building such program is not easy. Moreover, we can even further monitor lower-level system calls to profile a process.

For the performance concern of such security hyperviosr, we argue that due to the performance isolation of virtualized environment, the computation overhead is on the hypervisor rather than the guest. In our case, we can assign a dedicated virtual CPU (or physical CPU) to handle the security functions without interfering the execution of the guest. However, TEMU might lead to poor performance itself. We argue that it does not affect the correctness of the profiling and detection result. However, we consider that hardware-assistant-based implementation is more suitable for fast production in the future.

Although the PDA and ADA are more like behavior-based signature mechanisms, we anticipate that they can be used to perform behavior-based anomaly detection. As long as a normal behavior profile are learned or specified by the administrator, they can keep tracking the activities of the process and identify abnormal behavior . It can even more useful for preventing malicious activities or unknown attacks.

In this paper, we propose and implement a passive process activity analysis and an active fingerprinting methods for bot detection in virtualized environments. It can perform process tracking, tainted tracing, process profiling. They are the basis to provide secure guest for the cloud applications. Our system has the following benefits. (1) These methods are less intrusive than traditional host-based approach. (2) It can closely and more precisely monitor the behavior of bots. (3) The agents are implemented in the hypervisor. We detect the bots that access registry/file using API hooking technique without being aware of by the bot. (4) The cooperative passive and active detection methods provide proactive and effective bot detection.

The experiment results show that using bot behavior profiles learned, our passive detection agent can distinguish bot host with no false positive and no false negative. In addition, our active detection agent can detect real bot. Both detection method are workable and effective.

References

[1] M. A. Rajab *et al*., "A Multifaceted Approach to Understanding the Botnet Phenomenon," in *Proc. Internet Measurement Conference*, 2006.

[2] K. Rieck *et al*., "Learning and Classification of Malware Behavior", in *DIMVA*, 2008.

[3] C. Willems *et al*., "Toward Automated Dynamic Malware Analysis Using CWSandbox," in *Proc. IEEE Security & Privacy (S&P)*, 2007.

[4] G. Gu *et al*., "BotHunter: detecting malware infection through IDS-driven dialog correlation," in *Proc. USENIX Security Symposium*, 2007.

[5] G. Gu *et al*., "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *NDSS*, 2008.

[6] G. Gu *et al*., "BotMiner: clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *Proc. USENIX Security Symposium*, 2008.

[7] H. Yin *et al*., "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *CCS*, 2007.

[8] L. Liu *et al*., "BotTracer: Execution-Based Bot-Like Malware Detection," in *Proc. Int. Conf. on Information Security (ISC)*, 2008.

[9] D. E. Comer and J. C. Lin, "Probing TCP Implementations," in *Proc. USENIX Summer Conference*, 1994.

[10] J. M. Allen, "OS and Application Fingerprinting Techniques," SANS Institute, Tech. Rep. 2007.

[11] Testing for Web Application Fingerprint, https://owasp.org/index.php/Testing_for_Web_Application_Fingerprint_ (OWASP-IG-004)

[12] D. Brumley *et al*., "Automatically Identifying Trigger-Based Behavior in Malware," in *Botnet Detection*, 2008.

[13] D. Song *et al*., "Bitblaze: A new approach to computer security via binary analysis," in *Proc. Int. Conf. on Information Systems Security*, 2008.

[14] QEMU, http://wiki.qemu.org

[15] T. Garnkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *NDSS*, 2003.

[16] G. W. Dunlap *et al*., "Revirt: Enabling intrusion analysis through virtual- machine logging and replay," in *OSDI*, 2002.

[17] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[18] X. Jiang *et al*., "Stealthy Malware Detection through VMM-based "Out-of-the-Box" Semantic View Reconstruction," in *CCS*, 2007.

[19] Windows API List (Windows), http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx

[20] M. I. Sharif *et al*., "Secure in-VM Monitoring using Hardware Virtualization," in *CCS*, 2009.

[21] J. Pfoh *et al*., "A Formal Model for Virtual Machine Introspection," in *Proc. Workshop on Virtual Machine Security (VMSec)*, 2009.

[22] S.-W. Hsiao *et al*., "Cross-level behavioral analysis for robust early intrusion detection," in *Proc. IEEE Int. Conf. on Intelligence and Security Informatics (ISI)*, 2010.